Engineering Grammar-based Type Checking for Graph Rewriting Languages

Naoki Yamamoto Kazunori Ueda
Dept. of Computer Science and Engineering
Waseda University, Tokyo, Japan
{yamamoto,ueda}@ueda.info.waseda.ac.jp

The ability to handle evolving graph structures is important both for programming languages and modeling languages. Of various languages that adopt graphs as primary data structures, a graph rewriting language LMNtal provides features of both (concurrent) programming languages and modeling languages, and its implementation unifies ordinary program execution and model checking functionalities. Unlike pointer manipulation in imperative languages, LMNtal allows us to manipulate graph structures in such a way that the well-formedness of graphs is an invariant guaranteed by the language itself. However, since the shapes of graphs can be complex and diverse compared to algebraic data structures such as lists and trees, it is a non-obvious important task to formulate types of graphs to verify individual programs. With this motivation, this paper discusses LMNtal ShapeType, a type checking framework that applies the basic idea of Structured Gamma to a concrete graph rewriting language. Types are defined as generative grammars written as LMNtal rules, and type checking of LMNtal programs can be done by exploiting the model checking features of LMNtal itself. We gave a full implementation of type checking using the features of the LMNtal meta-interpreter.

1 Introduction

The ability to handle dynamically evolving graph structures is important both for programming languages and modeling languages. In programming, graphs appear both as *data structures* supporting efficient algorithms and as *process structures* exchanging messages through channels. In modeling, network structures can be found in many fields from the Internet to transportation, which can be modeled as graphs. Of various languages that adopt graphs as primary data structures, including GP 2 [1] and GROOVE [7], a graph rewriting language LMNtal [16] provides features of *programming* languages (including I/O and various other APIs) and those of *modeling* languages (including state space search). Its implementation, SLIM [8] (available from GitHub), provides ordinary program execution and parallel model checking (with 10^9 states) in a single framework. LMNtal allows us to handle data structures that cannot be succinctly modeled in functional languages. An example is a skip list [10] in Fig. 1(a), a linked list with additional edges skipping some nodes, which can be encoded into an LMNtal graph as in Fig. 1(b). Although LMNtal has simple syntax and semantics consisting of atoms, links and rewrite rules, it is Turing-complete and allow the encoding of various process calculi and the strong reduction of the λ-expressions in which both term structures and bound variables are represented as graphs [15].

Although LMNtal programs are pointer-safe in the sense that phenomena corresponding to dangling pointers and unintended aliasing in imperative languages never happen, it is possible that a graph with an unexpected shape is generated as a result of rewriting or computation gets stuck. For instance, see the two rewrite rules in Fig. 2. While the correct rule preserves the structure of a skip list, the incorrect one destroys the structure. Appropriate static type checking would detect such errors at compile time.

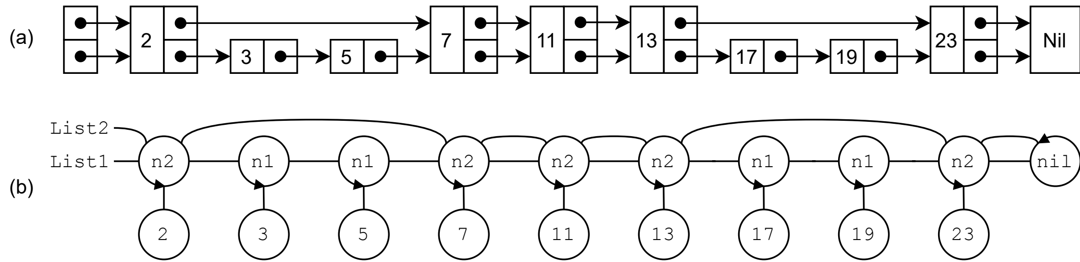


Figure 1: An example of skip list: (a) with pointers, (b) as an LMNtal graph. An arrowhead of each non-unary atom indicates the first argument and the ordering of arguments.

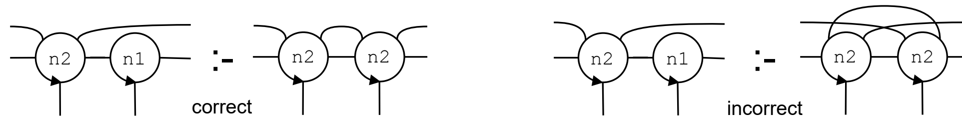


Figure 2: Example rules for skip list.

LMNtal ShapeType [19] is a type checking framework for LMNtal inspired by Shape types [5] which in turn is based on Structured Gamma [6]. Types are defined as generative grammars represented by rewrite rules of LMNtal. This makes it possible to do type checking by the non-deterministic execution of SLIM. Positioning a type description language as a sublanguage of the host language and making full use of the functionalities of the latter's implementation is a major strength of the present approach.

LMNtal ShapeType provides two basic type checking algorithms. One is *graph type checking* to check that a given graph is of the specified type. The other is *rule type checking* to check that a given rule will not destroy the structure of typed graphs.

The main contributions of this paper are threefold. First, we formalize LMNtal ShapeType addressing various subtleties. In previous studies, rule type checking had no explicit algorithm or correctness proof. Second, we expand the expressive power of the type checking. In order to handle constraints such as the balancing of trees and the number of elements an "express" link of a skip list can skip, we propose *extensive types* and *indexed types*[1] as new classes of graph types which are broader than context-free grammars. Finally, we propose a unified approach to check the type safety of *functional atoms*, i.e., atoms corresponding to functions (as opposed to data constructors) in other languages. In existing methods, type safety meant that each rewriting step would not destroy the structure of typed graphs. However, we often need to perform multi-step operations which may result in graphs of different types. We introduce a design pattern called functional atoms to check the type safety of such operations.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 introduces Flat LMNtal, the base language of the present work. Section 4 introduces LMNtal ShapeType and describes its type checking algorithms. Section 5 describes notable properties of LMNtal ShapeType. Section 6 introduces functional atoms and shows how to check their type safety. Section 7 discusses implementation.

[1] A variety of graph type definitions including the ones introduced in this paper can be found in `http://bit.ly/LMNtalShapeTypeEx`.

2 Related Work

There are several typing frameworks for graphs. Graph types [9] are a framework based on regular expressions. Structured Gamma [6] is a framework for graphs in which types are defined by production rules in context-free grammar. Shape types [5] are a subset of Structured Gamma and handle types which make the type checking algorithm complete. These methods ensure that graph structures expressed using pointers in C-like languages are consistent with the type definitions and that one-step operations on the graph structures will not affect the types of the structure. An algorithm proposed in [2] is able to handle shape-changing computation by specifying intermediate shapes, whereas our approach achieves the same objective using functional atoms. Besides, these methods are based on networks of pointers, and express graph edges by names (such as '`next`') and graph nodes by variables. This style is dual of our approach in which edges are expressed by α-convertible variables and nodes are expressed by atom names (such as '`cons`'). Although various formalisms of graph grammars [13] are well studied for decades, our technique, formulated in the framework of a practical concrete language, differs from those in many respects including the formulation of graphs and rewriting.

For our base language LMNtal, the method of [17] deals with 'microscopic' properties by giving capability types, which represent both polarities and sharing of (hyper)links, to local connection between nodes, while LMNtal ShapeType handles 'macroscopic' graph structures, i.e., shapes.

Separation logic [11] is well studied for reasoning about pointer structures, but the approach is different from ours in several respects: it deals with low-level languages and properties, while we consider graph structures formed by higher-level languages, abstracting pointers and heaps. Reasoning with separation logic uses proof assistants except for certain properties, while our objective is to pursue what properties can be established automatically using rather simple typing framework.

There are a lot of studies on handling quantitative properties in type systems [3][12]. Most of them enhance type systems with dependent types and employ decision procedures such as constraint solvers. While we pursue a somewhat close approach (Section 5.4), we also pursue an approach that does not employ numerical types for broader applications in mind (Section 5.1).

3 LMNtal: Graph Rewriting Language

In order to focus on the shape properties of graph rewriting in a concrete setting but without unnecessary complication, we consider a subset of LMNtal, called Flat LMNtal, which omits another structuring mechanism called *membranes*. This results in a significantly simpler fragment compared to the original setting [16]. We do not handle guards (for operations on built-in data types) or hyperlinks (for multi-point connectivity) either, but this core language still provides a powerful structuring mechanism. Hereinafter we simply call this subset LMNtal.

3.1 Syntax

The syntax of LMNtal is shown in Fig. 3. An LMNtal program is represented as a pair of a *graph* G (a multiset of atoms) and a *ruleset* R (a multiset of rewrite rules). This pair is called a *process*. An atom consists of an m-ary atom name p followed by m *totally ordered* link names $X_1,\ldots,X_m$. Names starting with capital letters are interpreted as link names, and others as atom names. The pair of the name and the arity of an atom are referred to as the *functor* of the atom and written as p/m. Atoms and links correspond to nodes and edges in graph theory, respectively. Unlike many other graph rewriting formalisms, graphs of LMNtal are defined in a syntax-directed manner and each atom has its own arity.

Process		::=	G, R	
Graph	G	::=	$\mathbf{0}$	(null)
		|	$p\,(X_1\,,\,\ldots\,,X_m)$	(atom, $m \geq 0$)
		|	$G\,,G$	(molecule)
Ruleset	R	::=	$\mathbf{0}$	(null)
		|	$[\mathit{RuleName}\,$@@$]\ G$:- G	(rule)
		|	$R\,,R$	(molecule)

Figure 3: Syntax of LMNtal.

In LMNtal, a multiset of atoms stands for an undirected multigraph, i.e., a graph that allows multi-edges and self-loops. For instance, a multiset of atoms `a(L1,F),b(L1,L2,L3,L4),` `c(L2,L5,L6,L6),d(L5,L3,L4)` stands for the undirected graph shown in Fig. 4.

A link name occurs at most twice in a graph. Link names occurring twice in a graph are called *local links* and those occurring once are called *free links* (a.k.a. half-edges) one of whose ends remains unconnected (or connected to the outside of the graph). Graphs without free links are *closed*. When we compose two graphs with a comma, we regard them to be α-converted as necessary to avoid collisions among local link names.

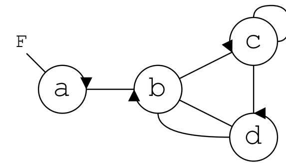


Figure 4: Pictorial representation of an LMNtal graph: `a(L1,F),b(L1,L2,L3,L4),` `c(L2,L5,L6,L6),d(L5,L3,L4)`

The following two abbreviations are allowed.

1. An atom written as another atom's argument is regarded as connected to the last argument of the outer atom, that is, $p\,(X_1\,,\ldots\,,X_{k-1}\,,q\,(Y_1\,,\ldots\,,Y_n)\,,X_{k+1}\,,\ldots\,,X_m)$ $(1 \leq k \leq m, 1 \leq n)$ is interpreted as $p\,(X_1\,,\ldots\,,X_{k-1}\,,L,X_{k+1}\,,\ldots\,,X_m)\,,q\,(Y_1\,,\ldots\,,Y_n\,,L)$ where L is a fresh link name. For instance, `a(b(c),d)` means `a(B,D),b(C,B),c(C),d(D)`.
2. An atom p`()` with no arguments may be written as p.

A *rule* (with an optional *RuleName*) describes rewriting of a subgraph to a subgraph. For instance, a rule `to(X,Y) :- from(Y,X)` rewrites a binary atom '`to`' to a binary atom '`from`' with its arguments swapped. For readability, rules may be written in a period-terminated form as well as in a comma-separated form. Because free links must not appear or disappear by rewriting, a link name in a rule must occur exactly twice.

3.2 Semantics

The semantics of LMNtal consists of *structural congruence* and *reduction relation*. We will introduce them in detail.

3.2.1 Structural Congruence

The syntax defined above does not (yet) characterize LMNtal graphs because the figure depicted in Fig. 4 corresponds to other syntactic representations of LMNtal graphs as well. We need to define an equivalence relation "$\equiv$", called *structural congruence*, to absorb syntactic variations. Structural

$$
\begin{array}{lrcll}
\text{(E1)} & \mathbf{0}\,,P & \equiv & P & \\
\text{(E2)} & P\,,Q & \equiv & Q\,,P & \\
\text{(E3)} & P\,,(Q\,,R) & \equiv & (P\,,Q)\,,R & \\
\text{(E4)} & P & \equiv & P[Y/X] & \text{(if } X \text{ is a local link of } P\text{)} \\
\text{(E5)} & P \equiv P' & \Rightarrow & P\,,Q \equiv P'\,,Q & \\
\text{(E7)} & X\text{=}X & \equiv & \mathbf{0} & \\
\text{(E8)} & X\text{=}Y & \equiv & Y\text{=}X & \\
\text{(E9)} & X\text{=}Y\,,P & \equiv & P[Y/X] & \text{(if } P \text{ is an atom and } X \text{ is a free link of } P\text{)}
\end{array}
$$

Figure 5: Structural congruence on LMNtal graphs.

$$
\text{(R1)}\ \frac{G_1 \xrightarrow{T\,:\text{-}\,U} G_1'}{G_1\,,G_2 \xrightarrow{T\,:\text{-}\,U} G_1'\,,G_2}
\qquad
\text{(R3)}\ \frac{G_2 \equiv G_1 \quad G_1 \xrightarrow{T\,:\text{-}\,U} G_1' \quad G_1' \equiv G_2'}{G_2 \xrightarrow{T\,:\text{-}\,U} G_2'}
\qquad
\text{(R6)}\ T \xrightarrow{T\,:\text{-}\,U} U
$$

Figure 6: Reduction relation on LMNtal graphs.

congruence is defined as the minimum equivalence relation satisfying the rules in Fig. 5. Structurally congruent LMNtal graphs are considered indistinguishable from each other. $P[Y/X]$ in (E4) and (E9) means to replace the link name X occurring in the graph P with the link name Y. Note that (E6) and (E10) in the original definition [16] are omitted because these are rules for membranes.

(E1)–(E3) characterize graph nodes as multisets. (E5) is a structural rule to make $\equiv$ a congruence. Both of them are standard rules found also in process algebra. (E4) is α-conversion of local link names[2]. Rules (E7) to (E9) are about the special binary atom = called a *connector*. An atom $=(X,Y)$, also written as $X=Y$, fuses two links X and Y. (E7) says that a self-closed link is regarded as a null graph, (E8) says that a connector is symmetric, and (E9) says that a connector may be absorbed or emitted by an atom. Connectors play an important role in writing rewrite rules such as

```
append(X,Y,Z), nil(X) :- Y=Z.
```

They play an important role in LMNtal ShapeType also.

3.2.2 Reduction Relation

"$\xrightarrow{T\,:\text{-}\,U}$", called a *reduction relation by the rule* T :- U, is a binary relation between two graphs, which describes the principal computation step in LMNtal. It is defined as the minimum binary relation satisfying the rules in Fig. 6. Note that (R2), (R4) and (R5) in [16] are omitted because these are for membranes.

The most important rule is (R6) which states that if there is a subgraph that matches the LHS of a rule, the subgraph can be rewritten into the RHS[3]. This definition can be naturally extended for rulesets, i.e., we say "G can transition (in one step) to G' by the ruleset R," written $G \xrightarrow{R} G'$, if $\exists r \in R.\ G \xrightarrow{r} G'$.

[2]The new link name Y must be "fresh" here; otherwise the graph $P[Y/X]$ violates the prerequisite that each link name can occur at most twice.

[3]For simplicity, we intentionally allow the case where T is null, which readily introduces divergence, though a legitimate implementation need not compile such rules.

In order to facilitate the formulation of our type checking method, the syntax and semantics described above separate graphs and rulesets, while in the standard definition they conjunctively form a process that may evolve autonomously.

3.3 Reverse Execution in LMNtal

Before we move on to the definition of LMNtal ShapeType, we introduce reverse execution in LMNtal. First we define the *inversion* of rules and rulesets.

Definition 1. The *inversion* r^{inv} of an LMNtal rule $r = T \texttt{ :- } U$ is defined by $r^{\mathrm{inv}} = U \texttt{ :- } T$. Likewise, the inversion R^{inv} of an LMNtal ruleset R is defined by $\forall r.\ r \in R \Leftrightarrow r^{\mathrm{inv}} \in R^{\mathrm{inv}}$.

It is important to note that an inverted rule is also a well-formed rule in LMNtal, and this plays a key role in our type checking method. Actually, reduction using an inverted rule is equivalent to following the reduction relation of the original rule in an opposite direction[4].

Proposition 1. For an LMNtal rule $r = T \texttt{ :- } U$, $G' \xrightarrow{r} G \Leftrightarrow G \xrightarrow{r^{\mathrm{inv}}} G'$.

Hence the reduction relation can be followed backward by executing the inversion of the LMNtal rule. This is called *reverse execution*.

4 LMNtal ShapeType

A typed language comes with a type description (sub)language, which is called LMNtal ShapeType in our setting. This section defines LMNtal ShapeType and describes its basic type checking algorithms. First, we give a formal definition. Following the terminology of formal language theory, LMNtal functors are hereinafter referred to as *symbols* also.

Definition 2. A *type* in LMNtal ShapeType (simply called *ShapeType*) is a triplet (S, P, N), where

- $S = t/m$ is a functor called the *start symbol*,
- P is a finite set of rules called *production rules*, and
- N is a finite set of functors called *nonterminal symbols*.

4.1 Syntax

The triplet of Definition 2 is written by the syntax of Fig. 7, where the start symbol S is given as an atom $p(X_1, \ldots, X_m)$[5], the production rules P as a ruleset R in Fig. 3, and nonterminal symbols N as a graph G in Fig. 3. *The LHS of each production rule must consist only of one or more nonterminal atoms which must not include connectors*. Abbreviations allowed for LMNtal atoms (Section 3.1) are also allowed.

ShapeType ::= `defshape` S `{` P `}` [`nonterminal {` N `}`]

Figure 7: Syntax of ShapeType.

[4] Proofs of all theorems, propositions and lemmas can be found in Appendix.

[5] We allow the case where $m = 0$, i.e., graphs with no roots, and non-connected graphs, for which our algorithms described in this section work as well.

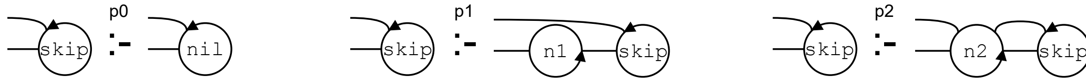


Figure 8: Production rules of the skip-list type `skip`.

By abuse of notation, functors in S and N are written as atoms in which link names are insignificant. Note also that S is always considered to be included in the set N of nonterminals (including the case where the definition of N is omitted).

An important feature of LMNtal ShapeType is that *a graph to be typed may have two or more roots (as in skip lists) whose order is significant*. Accordingly, when discussing whether a graph is of a specific type, a type is referred to as $t\,(L_1\,,\,\ldots\,,L_m)$ by explicitly mentioning roots. If the link names are not important, it is called type t/m (or type t if t is a unique start symbol name).

4.2 Semantics

We define the typing relation ":" using an auxiliary relation "$\lhd$" called the production relation. Hereinafter, a graph with free links $L_1,\ldots,L_m$ is written as $G[L_1,\ldots,L_m]$ and the set of all functors used in graph G is written as $\mathrm{Funct}(G)$.

Definition 3 (Production Relation)**.** For a type $(t/m,P,N)$, a graph $G[L_1,\ldots,L_m]$ is *generated by the type* $t\,(L_1\,,\,\ldots\,,L_m)$, written $G[L_1,\ldots,L_m] \lhd t\,(L_1\,,\,\ldots\,,L_m)$, iff $t\,(L_1\,,\,\ldots\,,L_m) \xrightarrow{P}{}^{*} G[L_1,\ldots,L_m]$.

Definition 4 (Typing Relation)**.** For a type $(t/m,P,N)$, a graph $G[L_1,\ldots,L_m]$ *has the type* $t\,(L_1\,,\,\ldots\,,L_m)$, written $G[L_1,\ldots,L_m] : t\,(L_1\,,\,\ldots\,,L_m)$, iff $G[L_1,\ldots,L_m] \lhd t\,(L_1\,,\,\ldots\,,L_m) \wedge \mathrm{Funct}(G[L_1,\ldots,L_m]) \cap N = \varnothing$.

Intuitively, only graphs to which the start symbol of type t can transition in zero or more steps by the production rules are said to be generated by type t, and only graphs with no nonterminal symbols are said to have the type t. In these definitions, free links of graphs and types are both explicitly written because their names and ordering are significant. For instance, given a type

```
defshape t(X,Y) { t(X,Y) :- a(X,Y) },
```

`a(X,Y)` $\lhd$ `t(X,Y)` holds but `a(Y,X)` $\lhd$ `t(X,Y)` does not hold.

The type of skip lists shown in Section 1 can be described as follows.

```
defshape skip(List2,List1){
  p0@@ skip(L2,L1) :- nil(L2,L1).
  p1@@ skip(L2,L1) :- n1(X1,L1), skip(L2,X1).
  p2@@ skip(L2,L1) :- n2(X1,X2,L2,L1), skip(X2,X1).
}
```

Note that elements in the skip list are omitted in order to focus on the structure of the graph, though it is possible to include elements and specify their types. The production rules can be visualized as in Fig. 8.

4.3 Graph Type Checking

Graph type checking is to check if an LMNtal graph X has a type t. The algorithm is shown in Fig. 9.

GCHECK is to check if the graph X has the type $(t/m,P,N)$. To check that X does not include a nonterminal symbol, we only have to check all the atoms because the number of atoms in X is finite. Then, it suffices to check that X is generated by $(t/m,P,N)$. GGCHECK checks that X can transition

// Checks that the LMNtal graph X has the type $(t/m,P,N)$
1: **function** GCheck($X,(t/m,P,N)$)
2: **if** $\exists f \in \mathrm{Funct}(X).\ f \in N$ **then return false**
3: **else return** GGCheck($X,(t/m,P,N),\varnothing$)
4: **end function**

// Checks that the LMNtal graph X can be generated by the type $(t/m,P,N)$
1: **function** GGCheck($X,(t/m,P,N),G$)
2: **if** $X \equiv t(L_1,\ldots,L_m)$ **then return true**
3: **for each** Y **s.t.** $X \xrightarrow{P^{\mathrm{inv}}} Y \land \nexists Y' \in G.\ Y \equiv Y'$ **do**
4: $G \leftarrow G \cup \{Y\}$
5: **if** GGCheck($Y,(t/m,P,N),G \cup \{X\}$) **then return true**
6: **end for**
7: **return false**
8: **end function**

Figure 9: Algorithm of graph type checking.

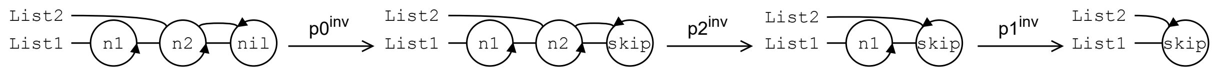


Figure 10: Reverse execution path of a skip list.

back to the start symbol by reverse execution with production rules P. For example, by reverse execution shown in Fig. 10, we can verify that the leftmost graph in Fig. 10 has the `skip` type. For a certain class of types that will be detailed in Section 5, the graph type checking algorithm satisfies soundness, completeness and termination for any rule.

4.4 Rule Type Checking

First, we define the type preservation property of an LMNtal rule r for the type t.

Definition 5 (Type Preservation)**.** Let r be an LMNtal rule, t be a type, and $L_1,\ldots,L_m$ be a sequence of links. We say that *r preserves t* iff

$$\forall G : t(L_1,\ldots,L_m).\quad G \xrightarrow{r} G' \Rightarrow G' : t(L_1,\ldots,L_m).$$

Checking the type preservation property is called *rule type checking*. The algorithm is shown in Fig. 11.

Intuitively, the algorithm checks if each generation path of L can be transformed to a generation path of R by structural induction on the production rules used last. RCheck ensures that both sides of the given rule consist only of the terminal symbols and then calls RCheckSub. RCheckSub recursively follows the production rule of t backwards from the LHS L, supplying a 'deficient' graph C to both sides (line 8), until the LHS reaches the start symbol (line 2) or a graph that appeared before (line 5). If it detects an LHS that appeared before, the algorithm backtracks to the point where the LHS appeared first. Positive return values are used to inform how many times the function should return by backtracking. Then Reduce verifies that the resulting graph augmented with the supplied graphs accumulated in the previous phase can transition to the RHS. Reduce shows this by reverse execution

```
   // Checks that the rule L :- R preserves the type (t/m,P,N)
 1: function RCheck(L :- R, (t/m,P,N))
 2:     return (Funct(L) ∪ Funct(R)) ∩ N = ∅ ∧ RCheckSub(L :- R, (t/m,P,N), []) = 0
 3: end function

   // Traverses the state space generated by reverse execution
   // (L :- R): current state, [v_1,...,v_n]: stack of visited states in depth first search
 1: function RCheckSub(L :- R, (t/m,P,N), [v_1,...,v_n])
 2:     if L consists only of one t/m atom then
 3:         if Reduce(R,L,P,∅) then return 0          ▷ Returns 0 if the rule preserves the type
 4:         else return −1                            ▷ Returns −1 if the rule may destroy the type
 5:     if ∃i s.t. 1 ≤ i ≤ n ∧ v_i = L_i :- R_i ∧ L_i ≡ L then
 6:         if Reduce(R_i,L_i,P,∅) then return n − i  ▷ Backtracks n − i steps when a cycle is detected
 7:         else return −1
 8:     for each (L',C) s.t. L,C --P^inv--> L' do     ▷ Explore all paths
 9:         S ← RCheckSub(L' :- (R,C), (t/m,P,N), [v_1,...,v_n,L :- R])
10:         if S > 0 then return S − 1                ▷ Continues backtracking
11:         if S = −1 then return −1  ▷ The rule may destroy the type if one or more calls return −1
12:     return 0
13: end function

   // Checks that there exists a reverse execution path from X to Y, i.e., X --P^inv-->* Y
   // G: visited graphs
 1: function Reduce(X,Y,P,G)
 2:     if X ≡ Y then return true
 3:     for each X' s.t. X --P^inv--> X' ∧ ∄X'' ∈ G. X'' ≡ X' do
 4:         G ← G ∪ {X'}
 5:         if Reduce(X',Y,P,G ∪ {X}) then return true
 6:     return false
 7: end function
```

Figure 11: Algorithm of rule type checking.

in order to prevent divergence. Finally, RCheckSub returns 0 if there is a state $L :- R$ on every path such that Reduce$(R,L,P,\varnothing)$ returns **true** and returns -1 otherwise. Note that each graph C enumerated in line 8 of RCheck must be the minimum one that enables matching with the RHS of rules and lets the reverse execution proceed.

For example, to verify that the rule in Fig. 12 preserves the `skip` type, we first go back from the LHS, supplying and accumulating necessary graphs (Fig. 13, upper), and then check if the resulting graph (`skip` in this example) can transition to the RHS with the supplied graphs (Fig. 13, lower).

This algorithm is inspired by that of Structured Gamma [6]. While the algorithm of Structured Gamma generates the entire state space first and the correspondence of free links in the supplied graphs was recorded and managed separately, our algorithm avoids this inconvenience by rewriting the target rule itself to generate the state space, using rewrite rules to represent individual states. This idea is similar to that of the sequent calculus in that it treats the pair of premise and conclusion as a single object.

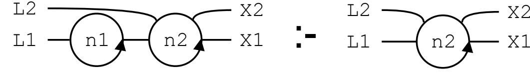


Figure 12: An example rule preserving the `skip` type.

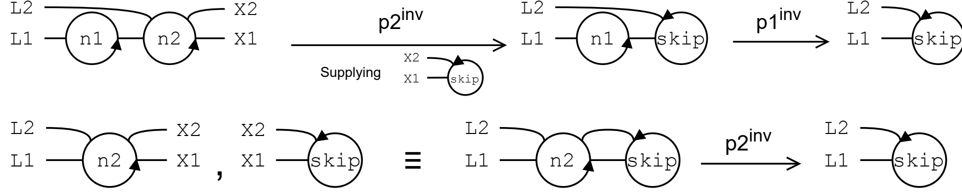


Figure 13: Reverse execution from the LHS (upper) and from the RHS (lower) of Fig. 12.

5 Properties of LMNtal ShapeType

This section describes some important properties of LMNtal ShapeType.

5.1 Extensive Types

The type checking algorithms introduced in [5] and [6] handle types with context-free production rules to ensure termination of graph type checking. *Context-free* types in our setting are defined as follows:

Definition 6. A production rule α :- β is *context-free* iff α consists of a single atom and β contains one or more atoms other than connectors. A type is context-free iff all of its production rules are context-free.

However, to ensure termination of graph type checking, it is sufficient if there is some measure for graphs whose values will not decrease by production rules, and this extension opens up versatile applications. Thus we propose *graph weighting* as a new measure for graphs.

Definition 7. For a type τ, a *weighting function* $w : \mathrm{Funct}(\tau) \to \mathbb{N} \setminus \{0\}$ weights each functor occurring in τ with a positive integer. As an exception, connectors' weight $w(=/2)$ must be zero because connectors can be arbitrarily absorbed or emitted by the structural congruence rule (E9).

Definition 8. Let w be a weighting function for a type τ and G be a graph which consists only of atoms with functors contained in $\mathrm{Funct}(\tau)$. The *weight of the graph* G, denoted $w(G)$, is defined by

$$w(G) = \sum_{p(L_1,\ldots,L_m)\in G} w(p/m).$$

Note that graph weighting generalizes the concept of the number of atoms.

Definition 9. A type $(t/m, P, N)$ is *extensive* iff $\exists w.\ \forall(\alpha \text{ :- } \beta) \in P.\ w(\alpha) \leq w(\beta)$.

Context-free types are obviously extensive. An extensive type corresponds to a length-increasing grammar in formal language theory, which is equivalent to a context-sensitive grammar. This extension is motivated by the need to handle a variety of types with non-context-free constraints as explained below[6].

For instance, the type of red-black trees can be defined as in Fig. 14. Here, the nonterminal symbol

[6] However, since LMNtal ShapeType handles graphs, care must be taken when discussing its expressive power. For instance, the list version of the typical context-sensitive language $\{a^n b^n c^n\}$ can be expressed by a simple context-free type in our setting.

```
defshape rbtree(R){
   s@@ rbtree(R) :- btree(nat,R).
  bz@@ btree(z,R) :- leaf(R).
  tz@@  tree(z,R) :- leaf(R).
  bs@@ btree(s(N),R) :- b(tree(N1),tree(N2),R), cp(N,N1,N2).
  ts@@  tree(s(N),R) :- b(tree(N1),tree(N2),R), cp(N,N1,N2).
   r@@  tree(N,R) :- r(btree(N1),btree(N2),R), cp(N,N1,N2).
  ns@@ nat(R) :- s(nat,R).
  nz@@ nat(R) :- z(R).
  cs@@ cp(s(N),N1,N2) :- cp(N,M1,M2), s(M1,N1), s(M2,N2).
  cz@@ cp(z,N1,N2) :- z(N1), z(N2).
} nonterminal {
  rbtree(R), btree(N,R), tree(N,R),
  cp(N,N1,N2), nat(R), s(N,R), z(R)
}
```

Figure 14: Type definition of red-black trees

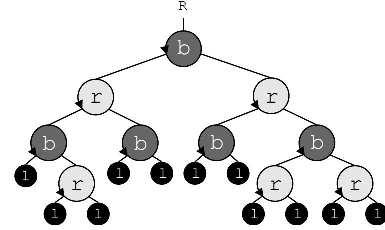


Figure 15: An example graph of the `rbtree` type (`l` stands for `leaf`).

`btree`/2 stands for a red-black tree with a black root and `tree`/2 stands for a red-black tree with a black or red root. In this definition, the black height of a tree to be generated is expressed using `z`/1 (zero) and `s`/2 (successor) atoms as X=`s(s(`...`s(z)`...`))` (as done in Prolog). The atom `cp`/3 copies and distributes the numeral connected to the first argument to the second and third arguments, with the production rules `cs` and `cz`. This definition expresses the balance of red-black trees by constraints on black heights distributed properly to subtrees. For instance, a graph in Fig. 15 has the `rbtree` type. This `rbtree` type is extensive with a weighting function $w(x)$ which returns 2 if $x = \texttt{leaf}/2$ and returns 1 otherwise. Note that it is easy to find an appropriate weighting since the constraint of extensive types (Def. 9) reduces to a system of linear inequalities.

The technique that employs `z`/1 and `s`/2 atoms can be used also for defining, e.g., skip lists with constraints on the number of stops that an "express" link can skip. Also, the type of the graph representation of λ-terms needs to manage a list (of unbounded length) of free links of subterms, which can be represented using an additional nonterminal symbol representing a list constructor.

For extensive types, the algorithm of graph type checking (Fig. 9) satisfies soundness, completeness, and termination.

Theorem 1 (Termination of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType τ, GCHECK(X,τ) terminates.

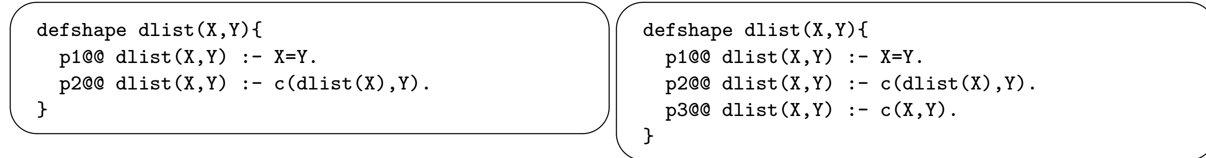
Theorem 2 (Soundness of graph type checking)**.** For any LMNtal graph X and a ShapeType $(t/m,P,N)$, if GCHECK$(X,(t/m,P,N))$ returns **true**, $X : t\,(L_1\,,\,\dots\,,L_m)$ holds.

Theorem 3 (Completeness of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType $(t/m,P,N)$, if $X : t\,(L_1\,,\,\dots\,,L_m)$ holds, GCHECK$(X,(t/m,P,N))$ returns **true**.

5.2 Production rules with connectors

The data structure called *difference lists* (*d-lists* for short), commonly used in Prolog programming since 1970's, represents a list with two variables representing the two ends. It enables constant-time concatenation by virtue of logic variables. Difference lists are extremely useful also in LMNtal programming where links are a special use of logic variables.

A type of d-lists can be written as in Fig. 16 (left). Since `p1` (for an empty d-list) has no atoms in the RHS except for a connector, there is no increasing weighting function for this type. The inversion of `p1`



```
defshape dlist(X,Y){
  p1@@ dlist(X,Y) :- X=Y.
  p2@@ dlist(X,Y) :- c(dlist(X),Y).
}
```

```
defshape dlist(X,Y){
  p1@@ dlist(X,Y) :- X=Y.
  p2@@ dlist(X,Y) :- c(dlist(X),Y).
  p3@@ dlist(X,Y) :- c(X,Y).
}
```

Figure 16: Type definition of d-lists (left) and its normalized definition (right).

is `X=Y :- dlist(X,Y)`, which matches *any* link, so reverse execution with this rule will not terminate. However, we cannot describe an *empty* d-list without such rules.

Let the target graph to be type-checked be G. If two free links of G are connected to each other, G cannot be expressed without connectors. A connector connecting two free links of G is called *global*. Now, we put a restriction that the inversion of a production rule fires only if all connectors in the RHS of the original production rule are global in G. With this restriction, a connector in the RHS of a production rule cannot match an arbitrary link.

Although this seems to decrease the flexibility of the types, the type description with this restriction retains the same expressiveness as the case where connectors can be freely used in the RHS of production rules. Actually, non-global connectors can be removed by a procedure like *the elimination of ε-rules* in the conversion of a context-free grammar to Chomsky normal form. Figure 16 (right) shows the result of such normalization.

Although we defined that connectors must have zero weight in Definition 7, global connectors may be weighted in the same way as ordinary functors. This is because only finitely many global connectors can appear in a graph and they cannot be absorbed or emitted. Note that a global connector cannot be a nonterminal symbol.

We go back to the `dlist` example. When all functors including the global connector are weighted 1, the production rules will not decrease the weight. Furthermore, graph type checking ensures that an empty d-list has the type `dlist` since `X=Y` can make reverse transition to `dlist(X,Y)` by the rule `p1`.

5.3 Rule type checking

The algorithm of rule type checking (Fig. 11) is sound in the following sense:

Theorem 4. For an LMNtal rule α :- β, a ShapeType $(t/m, P, N)$, and a sequence of links $L_1, \ldots, L_m$, if RCHECK(α :- $\beta, (t/m, P, N)$) returns **true**, the following formula (the type preservation property) holds:

$$\forall G : t(L_1, \ldots, L_m). \quad G \xrightarrow{\alpha \,:\text{-}\, \beta} G' \Rightarrow G' : t(L_1, \ldots, L_m)$$

For context-free types, the rule type checking algorithm terminates for any rule because the state space of our algorithm is the same as that of Structured Gamma [6] if we focus only on the LHS. However, it may not terminate for extensive types. For example, we can use the production rule `cz` of `rbtree` type (Fig. 14) backwards to transition from one `z` atom to `cp` and `z` atoms. This causes infinitely many `cp` atoms to be generated from a single `z` atom, so the rule type checking does not terminate.

As with other static type checking methods, completeness of rule type checking may not hold in general [6]. The proof of soundness assumes that there exists a transition path from the start symbol to a resulting graph of reverse execution from the LHS[7], but if there is no such path, the completeness does not hold. In this sense the algorithm is conservative.

[7] which is α_0 in the proof in Appendix.

```
defshape rbtree3(R){
  init@@ rbtree3(R) :- btree3(R).
  bb3@@ btree3(R) :- b(tree2,tree2,R).
  bb2@@ btree2(R) :- b(tree1,tree1,R).
  bb1@@ btree1(R) :- b(tree0,tree0,R).
  bl@@ btree0(R) :- leaf(R).
  tb2@@ tree2(R) :- b(tree1,tree1,R).
  tb1@@ tree1(R) :- b(tree0,tree0,R).
  tr2@@ tree2(R) :- r(btree2,btree2,R).
  tr1@@ tree1(R) :- r(btree1,btree1,R).
  tr0@@ tree0(R) :- r(btree0,btree0,R).
  tl@@ tree0(R) :- leaf(R).
} nonterminal {
  rbtree3(R), btree3(R), btree2(R),
  btree1(R), btree0(R),
  tree2(R), tree1(R), tree0(R)
}
```

```
defshape rbtree(R){
  init@@ rbtree(R) :- btree($n,R).
  bb@@ btree($n,R) :- $m = $n-1
    | b(tree($m),tree($m),R).
  bl@@ btree(0,R) :- leaf(R).
  tb@@ tree($n,R) :- $m = $n-1
    | b(tree($m),tree($m),R).
  tr@@ tree($n,R)
    :- r(btree($n),btree($n),R).
  tl@@ tree(0,R) :- leaf(R).
} nonterminal {
  rbtree(R), btree($n,R), tree($n,R)
}
```

Figure 17: Type definition of red-black trees as a context-free type (left) and as an indexed type (right).

5.4 Indexed Types

Types with numerical constraints, such as complete binary trees or (balanced) red-black trees of height n, or lists of length n, cannot be represented as context-free types. We could use extensive types (Section 5.1), for which the graph type checking works well, but the rule type checking for such types may not terminate.

On the other hand, types with constant numerical constraints, such as red-black trees with a height of exactly 3, can be represented by a context-free type as in Fig. 17 (left). The meanings of symbols in this definition are the same as those in Fig. 14 except that the names of nonterminal symbols are followed by indices representing the black height. In this definition, the production rules `bb3`–`bb1`, `tb2`–`tb1`, `tr2`–`tr0` are quite similar, respectively, so we can simplify them as in Fig. 17 (right), where we introduced a notation like '`$n`' to represent integer variables. This notation is borrowed from the *typed process context* [16], an extension of LMNtal. While the original typed process context is a mechanism to match any graph that satisfies the constraints specified in the *guard* (the part between `:-` and `|`), here we assume that all typed process contexts match only natural numbers[8]. This extension allows natural numbers and typed process contexts to appear in the LHS of the production rule in addition to one nonterminal symbol. Although this goes beyond the context-freeness assumption, the production rules are still essentially context-free in the sense that this natural number, which originally played the role of an index of a non-terminal symbol, can be regarded as an index of the non-terminal symbol. This idea of indexed nonterminal symbols was derived from indexed grammars in formal language theory.

Since these process contexts represent variables that may take any natural numbers, the rule type checking can cause state space explosion. Therefore, we ignore the difference of the indices of nonterminal symbols and consider them as the same state in the state space construction.

In this setting, the algorithm terminates because the state space of an indexed type is isomorphic to that of the type without indices. Since multiple states are represented by a single state, this may affect the completeness of type checking compared to the context-free version with a fixed size (Fig. 17, left). However, introducing indices is important not only for the simplicity of description but also for the

[8] In LMNtal, a number is represented as a unary atom with the atom name of that number.

expressiveness of the types since we cannot represent red-black trees of *any* height as a context-free type without indices.

6 Functional Atoms

A *functional atom* is a design pattern commonly used in LMNtal programming, which behaves like a function in functional languages. For instance, the atom append/3 with the following rules, depicted in Fig. 18 (upper), appends two lists like an append function in functional languages. The append/3 atom expects two lists connected to the first and second arguments as input, handles them by the rules, and returns the concatenated list through the third argument as in Fig. 18 (lower).

```
a1@@ R=append(c(L1),L2) :- R=c(append(L1,L2)).
a2@@ R=append(n,L)      :- R=L.
```

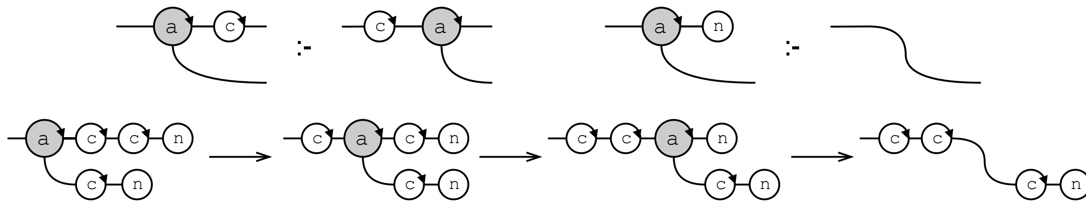


Figure 18: Rules of the append/3 atom (upper) and progress of computation (lower).

A functional atom may receive and/or return a value of a type with multiple roots. Consider the following functional atom t2l/3 (for tree-to-list):

```
t1@@ t2l(n(N,L,R),T,P) :- t2l(L,c(N,t2l(R,T)),P).
t2@@ t2l(l,T,P) :- T=P.
```

These rules are depicted in Fig. 19 (upper). The atom t2l/3 receives a binary tree, traverses it in-order, and returns a difference list as in Fig. 19 (lower).

By generalizing them, we formalize functional atoms as follows:

Definition 10 (Functional property)**.** For types $t_1,\dots,t_n$ and T, a graph F consisting of a single f/m atom, and a ruleset R consisting of rules each of which has just one f/m atom in the LHS, F is *functional* iff for all graphs $G_{t_1},\dots,G_{t_n}$ which have types $t_1,\dots,t_n$ respectively,

$$\forall G.\ \left(F\,,G_{t_1}\,,\,\dots\,,G_{t_n} \xrightarrow{R}{}^{*} G\right) \wedge \left(f/m \notin \mathrm{Funct}(G)\right) \ \Rightarrow\ G:T.$$

This property is written[9] as $t_1,\dots,t_n \vdash_R F : T$, where $t_1,\dots,t_n$ are called the *input types*, T the *output type*, and R the *functional ruleset for F*. With this notation, the properties of append and t2l are expressed as:

$$\texttt{list(L1),list(L2)} \vdash_{\{\texttt{a1,a2}\}} \texttt{append(L1,L2,R)} : \texttt{list(R)},$$
$$\texttt{tree(T)} \vdash_{\{\texttt{t1,t2}\}} \texttt{t2l(T,X,Y)} : \texttt{dlist(X,Y)}.$$

The functional property states that, when a functional atom F is given graphs with types $t_1,\dots,t_n$ as inputs, it returns a graph with type T upon termination.

The functional property is verified with rule type checking. Here, S_t, P_t, and N_t stand for the start symbol, the set of production rules, and the set of nonterminal symbols of type t, respectively.

[9] In this section, we assume that each type is written as an atom with the functor of its start symbol in order to clarify which argument of the functional atom is connected to which one of the types.

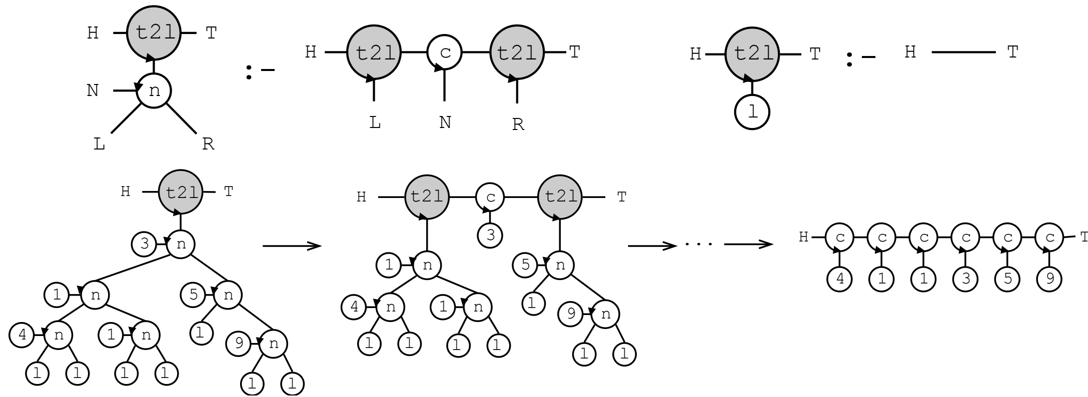


Figure 19: Rules of the `t2l/3` atom (upper) and progress of computation (lower).

Theorem 5. For a set of production rules $P = P_T \cup P_{t_1} \cup \cdots \cup P_{t_n} \cup \{T \texttt{ :- } F \texttt{,} t_1 \texttt{,} \ldots \texttt{,} t_n\}$ and a set of nonterminal symbols[10] $N = N_T \cup N_{t_1} \cup \cdots \cup N_{t_n}$, if every rule $r \in R$ preserves type (S_T, P, N), then $t_1, \ldots, t_n \vdash_R F : T$ holds.

For example, to check the functional property of `t2l`, we add a production rule `dlist(X,Y) :- t2l(T,X,Y),tree(T)` and the production rules of the type `tree` to the type `dlist` and and execute the rule type checking of `t1` and `t2`. Note that the functional property does not ensure that the computation with functional atoms will not get stuck because it assumes that the computation successfully terminates and no functional atom remains in the graph.

7 Implementation

While graph type checking can be easily done using the model checking features of SLIM, rule type checking is much harder to implement. Since the rule type checking algorithm requires special features to construct a state space in which states are represented as rewrite rules and states with the same LHSs are considered identical, we implemented this algorithm with the ideas and features of the LMNtal meta-interpreter [14].

To implement indexed types (Section 5.4), we need to perform operations on graphs containing numbers whose concrete values are not necessarily fixed but constrained. Since the current implementation of LMNtal cannot handle such indefinite numbers symbolically, we extended the data structure used in the above implementation on the LMNtal meta-interpreter to handle numerical constraints. These constraints belong to Presburger arithmetic, in which all formulas are decidable. We used the Z3 solver [4] as a backend to solve them. We have tested the implementation by typechecking reasonably complex operations on the graph structures exemplified in the paper, including the insertion operation into red-black trees (formulated using an indexed type) that requires rotation of trees, using functional atoms.

8 Conclusion and Future Work

We studied LMNtal ShapeType, a static type checking framework for a graph rewriting language LMNtal, and proposed extensions to enhance its expressiveness. First, we gave a formalization of the types

[10] Here we assume that there is no duplication among the nonterminal symbols.

and type checking algorithms for both graphs and rewrite rules, addressing various subtleties including those arising from connectors representing link fusion, a key construct in graph rewriting. Second, we proposed *extensive types* and *indexed types* as broader classes of graph types, which enabled us to handle constraints such as the balancing of trees. Third, we proposed a unified approach for checking the type safety of *functional atoms*, i.e., atoms interpreted as functions (as opposed to data constructors) in other languages, so that we could handle multi-step operations which might result in graphs of different types. The expressive power of our rather simple framework was demonstrated by defining a variety of graph types including skip lists with and without constraints, red-black trees, λ-terms represented as graphs, and rectangular grids with arbitrary or specified size, all available from the website given in Section 1.

For future work, it is important to expand the target language. We focused on Flat LMNtal without membranes or guards or hyperlinks [18], but they proved to be useful in computing with complicated graph structures. The challenge here is the handling of process contexts (a wildcard construct in graph matching) and guards (for expressing operations and constraints over built-in types), which is straightforward in forward execution but is challenging in backward execution both in theory and practice. Also, it is interesting to extend the framework to handle infinite graphs as input of functional atoms because the conception of LMNtal was the unified modeling of data structures and network of concurrent processes that evolve by exchanging messages. Functions modeled in LMNtal naturally allow concurrent and nonterminating execution, cooperating with each other by dataflow synchronization, and a unified modeling and reasoning framework of graph types with and without base cases is important future work. Finally, it is necessary to optimize the type checking algorithms. The algorithms worked well as a proof of concept and are simple enough to implement within the existing framework of LMNtal, but the exhaustive search can be too costly when target graphs or rules become complicated. Effective techniques for pruning search is an important topic of future work.

Acknowledgments. The authors would like to thank anonymous reviewers for their useful comments. This work was partially supported by Grant-in-Aid for Scientific Research (B) JP18H03223, JSPS, Japan, and Waseda University Grant for Special Research Projects (2021C-142).

References

[1] Christopher Bak (2015): *GP 2: efficient implementation of a graph programming language*. Ph.D. thesis, Department of Computer Science, The University of York. Available at `https://etheses.whiterose.ac.uk/12586/`.

[2] Adam Bakewell, Detlef Plump & Colin Runciman (2004): *Checking the Shape Safety of Pointer Manipulations*. In: *Proc. RelMiCS 2003*, *LNCS* 3051, Springer, pp. 48–61. Available at `https://doi.org/10.1007/978-3-540-24771-5_5`.

[3] W.-N. Chin & S.-C. Khoo (2001): *Calculating sized types*. *Higher-Order and Symbolic Computation* 14(2–3), pp. 260–300. Available at `https://doi.org/10.1145/328690.328893`.

[4] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An efficient SMT solver*. In: *Proc. TACAS 2008*, *LNCS* 4963, Springer, pp. 337–340. Available at `https://doi.org/10.1007/978-3-540-78800-3_24`.

[5] Pascal Fradet & Daniel Le Métayer (1997): *Shape types*. In: *Proc. POPL'97*, ACM, pp. 27–39. Available at `https://doi.org/10.1145/263699.263706`.

[6] Pascal Fradet & Daniel Le Métayer (1998): *Structured Gamma*. *Science of Computer Programming* 31(2), pp. 263–289. Available at `https://doi.org/10.1016/S0167-6423(97)00023-3`.

[7] A.H. Ghamarian, M. de Mol, A. Rensink, E. Zambon & M. Zimakova (2012): *Modelling and analysis using GROOVE*. *STTT* 14(1), pp. 15–40. Available at `https://doi.org/10.1007/s10009-011-0186-x`.

[8] Masato Gocho, Taisuke Hori & Kazunori Ueda (2011): *Evolution of the LMNtal Runtime to a Parallel Model Checker*. Computer Software 28(4), pp. 4_137–4_157. Available at `https://doi.org/10.11309/jssst.28.4_137`.

[9] Nils Klarlund & Michael I. Schwartzbach (1993): *Graph Types*. In: Proc. POPL'93, ACM, pp. 196–205. Available at `https://doi.org/10.1145/158511.158628`.

[10] William Pugh (1990): *Skip lists: A probabilistic alternative to balanced trees*. Commun. ACM 33(6), pp. 668–676. Available at `https://doi.org/10.1145/78973.78977`.

[11] J.C. Reynolds (2002): *Separation logic: a logic for shared mutable data structures*. In: Proc. LICS 2002, IEEE, pp. 55–74. Available at `https://doi.org/10.1109/LICS.2002.1029817`.

[12] Patrick M. Rondon, Ming Kawaguci & Ranjit Jhala (2008): *Liquid Types*. In: Proc. PLDI 2008, ACM, pp. 159–169. Available at `https://doi.org/10.1145/1375581.1375602`.

[13] Grzegorz Rozenberg (1997): *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific. Available at `https://doi.org/10.1142/3303`.

[14] Yutaro Tsunekawa, Taichi Tomioka & Kazunori Ueda (2018): *Implementation of LMNtal Model Checkers: A Metaprogramming Approach*. Journal of Object Technology 17(1). Available at `https://doi.org/10.5381/jot.2018.17.1.a1`.

[15] Kazunori Ueda (2008): *Encoding the Pure Lambda Calculus into Hierarchical Graph Rewriting*. In: Proc. RTA2008, LNCS 5117, Springer, pp. 392–408. Available at `https://doi.org/10.1007/978-3-540-70590-1_27`.

[16] Kazunori Ueda (2009): *LMNtal as a hierarchical logic programming language*. Theoretical Computer Science 410(46), pp. 4784–4800. Available at `https://doi.org/10.1016/j.tcs.2009.07.043`.

[17] Kazunori Ueda (2014): *Towards a Substrate Framework of Computation*. In: Concurrent Objects and Beyond, LNCS 8665, Springer, pp. 341–366. Available at `https://doi.org/10.1007/978-3-662-44471-9_15`.

[18] Kazunori Ueda & Seiji Ogawa (2012): *HyperLMNtal: An Extension of a Hierarchical Graph Rewriting Model*. KI - Künstliche Intelligenz 26(1), pp. 27–36. Available at `https://doi.org/10.1007/s13218-011-0162-3`.

[19] Yusuke Yoshimoto & Kazunori Ueda (2015): *Static type checking in graph rewriting system*. In: Proceedings of the 32nd JSSST Annual Conference. Available at `http://jssst.or.jp/files/user/taikai/2015/PPL/ppl3-3.pdf`.

Appendix

Hereinafter, for a set of functors F, $\mathrm{Graph}(F)$ stands for the set of every graph G s.t. $\mathrm{Funct}(G) \subseteq F$.

Proposition 1. For LMNtal rule $r = T \text{ :- } U,\ G' \xrightarrow{r} G \Leftrightarrow G \xrightarrow{r^{\mathrm{inv}}} G'$

Proof. ($\Rightarrow$) We will prove by structural induction on the last-used reduction relation rule.

- Case (R1): We assume that $G' = P, Q,\ G = P', Q$ and $P \xrightarrow{r} P'$. By the induction hypothesis, $P' \xrightarrow{r^{\mathrm{inv}}} P$. By (R1), $P', Q \xrightarrow{r^{\mathrm{inv}}} P, Q$. Therefore, $G \xrightarrow{r^{\mathrm{inv}}} G'$.
- Case (R3): We assume that $G' \equiv P,\ P' \equiv G,\ P \xrightarrow{r} P'$. By the induction hypothesis, $P' \xrightarrow{r^{\mathrm{inv}}} P$. By (R3) and $G' \equiv P,\ P' \equiv G$, we obtain $G \xrightarrow{r^{\mathrm{inv}}} G'$.
- Case (R6) is self-evident by (R6).

Hence $G' \xrightarrow{r} G \Rightarrow G \xrightarrow{r^{\mathrm{inv}}} G'$.

($\Leftarrow$) follows from $(r^{\mathrm{inv}})^{\mathrm{inv}} = r$ and ($\Rightarrow$). □

Lemma 1. For an extensive ShapeType $\tau = (t/m, P, N)$, its weighting function w, and an LMNtal graph $G, G' \in \mathrm{Graph}(\mathrm{Funct}(\tau))$,

$$G' \xrightarrow{P} G \Rightarrow w(G) \geq w(G')$$

Proof. By $G' \xrightarrow{P} G$, let $p = \alpha \text{ :- } \beta$ be a rule s.t. $G' \xrightarrow{p} G$. We will show $w(G) \geq w(G')$ by structural induction on the last-used reduction relation rule.

- Case (R1): We assume that $G' = G'_1, G_2,\ G = G_1, G_2$ and $G'_1 \xrightarrow{p} G_1$. Then $w(G_1) \geq w(G'_1)$ by the induction hypothesis. We have $w(G') = w(G'_1) + w(G_2),\ w(G) = w(G_1) + w(G_2)$, therefore $w(G) \geq w(G')$ holds.
- Case (R3): We assume that $G'_1 \equiv G',\ G \equiv G_1,\ G'_1 \xrightarrow{p} G_1$. Then $w(G_1) \geq w(G'_1)$ by the induction hypothesis. We have $w(G_1) = w(G),\ w(G'_1) = w(G')$, therefore $w(G) \geq w(G')$ holds.
- Case (R6): We assume that $G' = \alpha,\ G = \beta$. Since the type $(t/m, P, N)$ is extensive, we have $w(\alpha) \leq w(\beta)$ i.e. $w(G) \geq w(G')$. □

Lemma 2. For an extensive ShapeType $\tau = (t/m, P, N)$, its weighting function w, a non-negative integer n, and a finite set of links L, the number of LMNtal graph G satisfying following formula is *finite* regarding structurally congruent graphs as the same.

$$G \in \mathrm{Graph}(\mathrm{Funct}(\tau)) \wedge w(G) = n \wedge \mathrm{FLink}(G) = L$$

Proof. The number of functors occurring in G is finite and also the number of atoms (excluding non-global connectors) occurring in G is less than n because of $w(G) = n$. Since the number of graphs consisting of finite kinds of functors and finite atoms is finite, the number of G is finite. □

Lemma 3. For any LMNtal graph X, an extensive ShapeType $(t/m, P, N)$, and a finite set of LMNtal graphs G, $\mathrm{GGC{\scriptstyle HECK}}(X, (t/m, P, N, G))$ terminates.

Proof. Let the weighting function of type $(t/m, P, N)$ be w. Y given to the first argument of recursive call at line 5 of GGCHECK satisfies $Y \xrightarrow{P}{}^{*} X$. By Lemma 1, $w(X) \geq w(Y)$. By this and Lemma 2, the number of possible Y is finite (regarding structurally congruent graphs are the same). Also, it is verified at line 3 of GGCHECK that structurally congruent graphs cannot be given again to the argument of recursive calls, so that recursive calls occur finite times at most. Therefore GGCHECK$(X, (t/m, P, N), G)$ terminates. □

Theorem 1 (Termination of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType τ, GCHECK(X, τ) terminates.

Proof. This follows by Lemma 3 and the fact that Funct(X) and N are finite. □

Lemma 4. For any LMNtal graph X, a ShapeType $(t/m, P, N)$, and a finite set of LMNtal graphs G, if GGCHECK$(X, (t/m, P, N), G)$ returns **true**, $X \lhd t(L_1, \dots, L_m)$ holds.

Proof. We will show by induction on the maximum number n of times of recursive calls (i.e. the depth of recursion) of GGCHECK.

- If $n = 0$, **true** is returned at line 2. Also, we have $X \equiv t(L_1, \dots, L_m)$. Then it is clear by the definition that $X \lhd t(L_1, \dots, L_m)$.
- If $n = k+1$ $(k \geq 0)$, **true** is returned at line 5 since recursive calls occur one or more times. We have $Y \xrightarrow{P} X$ by the line 3 and $Y \lhd t(L_1, \dots, L_m)$ by the induction hypothesis. Then $t(L_1, \dots, L_m) \xrightarrow{P}{}^{*} Y$ follows by the definition of production relations. By $Y \xrightarrow{P} X$, we have $Y \xrightarrow{P} X$, so that $t(L_1, \dots, L_m) \xrightarrow{P}{}^{*} X$. Therefore $X \lhd t(L_1, \dots, L_m)$ holds. □

Theorem 2 (Soundness of graph type checking)**.** For any LMNtal graph X and a ShapeType $(t/m, P, N)$, if GCHECK$(X, (t/m, P, N))$ returns **true**, $X : t(L_1, \dots, L_m)$ holds.

Proof. GCHECK$(X, (t/m, P, N))$ returns **true** only when $\exists f \in \text{Funct}(X).\ f \in N$ does not hold and then it just returns the returned value from GGCHECK$(X, (t/m, P, N), \varnothing)$, so that GGCHECK$(X, (t/m, P, N), \varnothing)$ returns **true**. By Lemma 4, we have $X \lhd t(L_1, \dots, L_m)$. Since $\neg\exists f \in \text{Funct}(X).\ f \in N$ holds, we have $\forall f \in \text{Funct}(X).\ f \notin N$. Therefore $\text{Funct}(X) \cap N = \varnothing$ holds. Hence we have $X : t(L_1, \dots, L_m)$. □

Lemma 5. For an LMNtal graph X and a ShapeType $(t/m, P, N)$, if $X \lhd t(L_1, \dots, L_m)$, GGCHECK$(X, (t/m, P, N), \varnothing)$ returns **true**.

Proof. By $X \lhd t(L_1, \dots, L_m)$, for certain $X_0, \dots, X_n$ $(n \geq 0)$, the following holds:

$$t(L_1, \dots, L_m) = X_n \xrightarrow{P} \dots \xrightarrow{P} X_1 \xrightarrow{P} X_0 = X$$

Note that X_i is not the start symbol for every i $(i < n)$ and $i \neq j \Rightarrow X_i \not\equiv X_j$ holds (i.e. no loops in the path). Consider the case when GGCHECK$(Y_i, (t/m, P, N), G_i)$ is called for i $(i < n)$, Y_i s.t. $Y_i \equiv X_i$, and a certain G_i. Since X_i is not the start symbol, Y_i is also not the start symbol, so that the condition of the if statement at line 2 does not hold. Then we have $X_{i+1} \xrightarrow{P} Y_i$ by $X_{i+1} \xrightarrow{P} X_i$ and (R3).

- If $\nexists Y_{i+1} \in G_i.\ X_{i+1} \equiv Y_{i+1}$, the for-loop from the line 3 is executed for $Y \leftarrow X_{i+1}$, and then GGCHECK$(X_{i+1}, (t/m, P, N), G_{i+1})$ is called for a certain G_{i+1} at line 5.
- If $\exists Y_{i+1} \in G_i.\ X_{i+1} \equiv Y_{i+1}$, GGCHECK$(Y_{i+1}, (t/m, P, N), G_{i+1})$ has been called for a certain G_{i+1} elsewhere.

Therefore GGCHECK($Y_{i+1}, (t/m, P, N), G_{i+1}$) is called for Y_{i+1} s.t. $Y_{i+1} \equiv X_{i+1}$ and a certain G_{i+1} somewhere in the recursive calls.

From the above reasons, when GGCHECK($X, (t/m, P, N), \varnothing$) is called, GGCHECK($Y_n, (t/m, P, N), G_n$) is also called for Y_n s.t. $Y_n \equiv X_n$ and a certain G_n. This call returns **true** at line 2 because of $Y_n \equiv X_n = t\,(L_1\,,\,\dots\,,L_m)$.

Therefore GGCHECK($X, (t/m, P, N), \varnothing$) returns **true** since GGCHECK returns **true** if one or more recursive calls in it return **true**. □

Theorem 3 (Completeness of graph type checking)**.** For any LMNtal graph X and an extensive ShapeType $(t/m, P, N)$, if $X : t\,(L_1\,,\,\dots\,,L_m)$ holds, GCHECK($X, (t/m, P, N)$) returns **true**.

Proof. By $X : t\,(L_1\,,\,\dots\,,L_m)$, we have $X \lhd t\,(L_1\,,\,\dots\,,L_m)$ and $\mathrm{Funct}(X) \cap N = \varnothing$. Therefore $\forall f \in Funct(X).\ f \notin X$, so that the condition of the if statement at line 2 of GCHECK does not hold. Then GCHECK($X, (t/m, P, N)$) just returns the returned value from GGCHECK($X, (t/m, P, N), \varnothing$). By $X \lhd t\,(L_1\,,\,\dots\,,L_m)$ and Lemma 5, GGCHECK($X, (t/m, P, N), G$) returns **true**. Therefore GCHECK($X, (t/m, P, N)$) returns **true**. □

Definition 11. A transition relation $\mathscr{T}$ between LMNtal rules α_1 :- β_1 and α_2 :- β_2 is defined as follows:

$$\begin{aligned} &\alpha_1 \text{ :- } \beta_1 \xrightarrow{\mathscr{T}} \alpha_2 \text{ :- } \beta_2 \\ &\quad \text{iff} \quad \exists \alpha_p \text{ :- } \beta_p \in P.\ \exists \gamma, \gamma'. \\ &\qquad\quad \alpha_2 \xrightarrow{\alpha_p \text{ :- } \beta_p} \alpha_1\,,\gamma \wedge \beta_2 \equiv \beta_1\,,\gamma \\ &\qquad\qquad \wedge\ \beta_p \equiv \gamma, \gamma' \wedge \gamma' \not\equiv \mathbf{0} \end{aligned}$$

Next, we define a labeling function $\mathscr{L} : \mathscr{W} \to 2^{\{\mathtt{s},\mathtt{r}\}}$ as follows, where $\mathscr{W}$ is the whole set of LMNtal rules:

$$\begin{aligned} \mathtt{s} \in \mathscr{L}(\alpha \text{ :- } \beta) \quad &\text{iff} \quad \alpha \equiv T \\ \mathtt{r} \in \mathscr{L}(\alpha \text{ :- } \beta) \quad &\text{iff} \quad \alpha \xrightarrow{P}{}^{*} \beta \end{aligned}$$

If $\mathtt{r} \in \mathscr{L}(\alpha \text{ :- } \beta)$, we say α :- β is *reducible*. Then we consider a Kripke structure $\mathscr{S} = (\mathscr{W}, \mathscr{T}, \mathscr{L})$ which represents the state space of the rule type checking algorithm.

Lemma 6. If α :- β is reducible and $\alpha \text{ :- } \beta \xrightarrow{\mathscr{T}} \alpha' \text{ :- } \beta'$, then α' :- β' is reducible.

Proof. By the assumption, we have $\alpha \xrightarrow{P}{}^{*} \beta$. By the definition of $\mathscr{T}$, we have $\exists \gamma.\ \alpha' \xrightarrow{P} \alpha\,,\gamma\beta' \equiv \beta\,,\gamma$. Then we have $\alpha' \xrightarrow{P} \alpha\,,\gamma \xrightarrow{P}{}^{*} \beta\,,\gamma \equiv \beta'$, and $\alpha' \xrightarrow{P}{}^{*} \beta'$ holds. □

Lemma 7. If $P\,,Q \equiv R\,,S$ holds, $P \equiv A_1\,,A_2,\ Q \equiv A_3\,,A_4,\ R \equiv A_1\,,A_3,\ S \equiv A_2\,,A_4$ holds for certain graphs A_1, A_2, A_3, A_4.

Proof. This follows by the rules of structural congruence. □

Lemma 8. If $P \xrightarrow{\alpha \text{ :- } \beta} Q$ holds, there exists a graph C that satisfies $P \equiv C\,,\alpha,\ Q \equiv C\,,\beta$.

Proof. This follows by the rules of structural congruence and reduction relation. □

Lemma 9. If $X \xrightarrow{p} Y \equiv \alpha\,,C$ ($p = \alpha_p \text{ :- } \beta_p \in P$) holds, one of the followings holds:

- $\forall \beta.\ \exists \alpha', \beta', C_1, C_2.\ \alpha \,{:}\!-\, \beta \xrightarrow{\mathscr{T}} \alpha' \,{:}\!-\, \beta' \ \wedge\ C \equiv C_1, C_2 \wedge X \equiv \alpha', C_2 \wedge \beta' \equiv \beta, C_1$
- $\exists C'.\ X \equiv \alpha, C' \wedge C' \xrightarrow{p} C$

Proof. By $X \xrightarrow{p} Y$ and Lemma 8, there exists C_p s.t. $X \equiv \alpha_p, C_p$, $Y \equiv \beta_p, C_p$. Then we have $Y \equiv \beta_p, C_p \equiv \alpha, C$ and, by Lemma 7, there exist C_1, C_2, C_3, C_4 s.t. $C \equiv C_1, C_2,\ \alpha \equiv C_3, C_4,\ \beta_p \equiv C_1, C_3,\ C_p \equiv C_2, C_4$.

Case 1: $C_3 \not\equiv \mathbf{0}$

Let $\alpha' = \alpha_p, C_4,\ \beta' = \beta, C_1$. Then we have $\alpha' \equiv \alpha_p, C_4 \xrightarrow{p} \beta_p, C_4 \equiv C_1, C_3, C_4 \equiv \alpha, C_1$. Here we consider C_1, C_3 as γ, γ' in the definition $\mathscr{T}$ respectively, and we have $\alpha \,{:}\!-\, \beta \xrightarrow{\mathscr{T}} \alpha' \,{:}\!-\, \beta'$. Also, we have $X \equiv \alpha_p, C_p \equiv \alpha_p, C_2, C_4 \equiv \alpha', C_2$.

Case 2: $C_3 \equiv \mathbf{0}$

By $C_1 \equiv \beta_p$, we have $C \equiv \beta_p, C_2$. Therefore $\alpha_p, C_2 \xrightarrow{p} C$ holds. Here we consider C' s.t. $C' \equiv \alpha_p, C_2$, then $C' \xrightarrow{p} C$ holds. Besides, by $\alpha \equiv C_4$, we have $C_p \equiv C_2, \alpha$. Then we have $X \equiv \alpha_p, C_p \equiv \alpha_p, C_2, \alpha \equiv \alpha, C'$. □

Lemma 10. If $\mathscr{S}, r \models \neg \mathtt{s}\, \mathsf{W}\, \mathtt{r}$ holds, r preserves the type T.

Proof. We assume $G \lhd T$, $G \xrightarrow{r} G'$, and $\mathscr{S}, r \models \neg \mathtt{s}\, \mathsf{W}\, \mathtt{r}$, and we will prove $G' \lhd T$.

By $G \lhd T$, we have $\forall i < n.\ X_{i+1} \xrightarrow{p_i} X_i$ for a certain non-negative integer n, graphs $X_0, \ldots, X_n$ ($X_0 = G, X_n = T$), and $p_0, \ldots, p_{n-1} \in P$. By $G \xrightarrow{r} G'$, there exists $C \in \mathscr{G}(N \cup \Sigma)$ s.t. $G \equiv \alpha, C,\ G' \equiv \beta, C$ where $r = \alpha \,{:}\!-\, \beta$.

Next, we will show that, if $X_i \equiv \alpha_i, C_i$ holds, there exist $\alpha_{i+1}, \beta_{i+1}, C_{i+1}$ such that:

$$\alpha_i \,{:}\!-\, \beta_i \xrightarrow{\mathscr{T}}{}^{*} \alpha_{i+1} \,{:}\!-\, \beta_{i+1} \ \wedge\ X_{i+1} \equiv \alpha_{i+1}, C_{i+1} \ \wedge\ \beta_{i+1}, C_{i+1} \xrightarrow{P}{}^{*} \beta_i, C_i$$

By $X_{i+1} \xrightarrow{p_i} X_i$ and Lemma 9, one of the following holds:

1. $\exists \alpha_{i+1}, \beta_{i+1}, C'_i, C_{i+1}.\ \alpha_i \,{:}\!-\, \beta_i \xrightarrow{\mathscr{T}} \alpha_{i+1} \,{:}\!-\, \beta_{i+1} \wedge C_i \equiv C'_i, C_{i+1} \wedge X_{i+1} \equiv \alpha_{i+1}, C_{i+1} \wedge \beta_{i+1} \equiv \beta_i, C'_i$
2. $\exists C_{i+1}.\ X_{i+1} \equiv \alpha_i, C_{i+1} \wedge C_{i+1} \xrightarrow{p_i} C_i$

If 1. holds, it is obvious since we have $\beta_{i+1}, C_{i+1} \equiv \beta_i, C'_i, C_{i+1} \equiv \beta_i, C_i$. On the other hand, if 2. holds, it is obvious when we consider $\alpha_{i+1} = \alpha_i,\ \beta_{i+1} = \beta_i$.

Thus, there exist α_n, β_n, C_n s.t. $\alpha \,{:}\!-\, \beta \xrightarrow{\mathscr{T}}{}^{*} \alpha_n \,{:}\!-\, \beta_n$, $T \equiv \alpha_n, C_n$, and $\beta_n, C_n \xrightarrow{P}{}^{*} \beta, C$. Since T consists only of one atom of the start symbol (with no self loops), we have $T \equiv \alpha_n$. Then $\mathtt{s} \in \mathscr{L}(\alpha_n \,{:}\!-\, \beta_n)$ holds.

By $r \xrightarrow{\mathscr{T}}{}^{*} \alpha_n \,{:}\!-\, \beta_n$, $\mathscr{S}, r \models \neg \mathtt{s}\, \mathsf{W}\, \mathtt{r}$, and Lemma 6, we also have $\mathtt{r} \in \mathscr{L}(\alpha_n \,{:}\!-\, \beta_n)$. Therefore we have $\alpha_n \xrightarrow{P}{}^{*} \beta_n$. Thus we have $T \equiv \alpha_n \xrightarrow{P}{}^{*} \beta_n \xrightarrow{P}{}^{*} \beta, C \equiv G'$, that is, $G' \lhd T$. □

Theorem 4 (Soundness of rule type checking)**.** For an LMNtal rule $\alpha \,{:}\!-\, \beta$, a ShapeType $(t/m, P, N)$, and a sequence of links $L_1, \ldots, L_m$, if RCHECK$(\alpha \,{:}\!-\, \beta, (t/m, P, N))$ returns **true**, the following formula (the rule preserving property) holds:

$$\forall G : t(L_1, \ldots, L_m).\quad G \xrightarrow{\alpha \,{:}\!-\, \beta} G' \Rightarrow G' : t(L_1, \ldots, L_m)$$

Proof. Since RCHECK$(\alpha \,{:}\!-\, \beta, (t/m, P, N))$ returns **true**, on all the paths from the target rule to the start symbol, there exists a state $L \,{:}\!-\, R$ such that REDUCE$(R, L, P, \varnothing)$ returns **true**. Therefore we have $\mathscr{S}, r \models \neg \mathtt{s}\, \mathsf{W}\, \mathtt{r}$, and the rule preserving property holds by Lemma 10. □

Theorem 5. For a set of production rules $P = P_T \cup P_{t_1} \cup \cdots \cup P_{t_n} \cup \{T \text{ :- } F\,, t_1\,, \ldots\,, t_n\}$, and a set of nonterminal symbols $N = N_T \cup N_{t_1} \cup \cdots \cup N_{t_n}$, if every rule $r \in R$ preserves type (S_T, P, N), $t_1, \ldots, t_n \vdash_R F : T$ holds.

Proof. $F\,, G_{t_1}\,, \ldots\,, G_{t_n}$ has the type (S_T, P, N) because P includes the production rule $T \text{ :- } F\,, t_1\,, \ldots\,, t_n$ and T is the start symbol. Let G be a graph s.t. $F\,, G_{t_1}\,, \ldots\,, G_{t_n} \xrightarrow{R}{}^{*} G$. Then G has the type (S_T, P, N) because every rule $r \in R$ preserves the type. Here we assume that G contains no f/m atoms. By G : (S_T, P, N), there exists a production path s.t. $S_T \xrightarrow{P}{}^{*} G$. Since G contains no f/m atoms, the production rule $T \text{ :- } F\,, t_1\,, \ldots\,, t_n$ has not been applied in the production path. Also the nonterminal symbols of the types $t_1, \ldots, t_n$ do not appear in the production path because they can appear only after the production rule $T \text{ :- } F\,, t_1\,, \ldots\,, t_n$ is applied. Therefore the production rules of the types $t_1, \ldots, t_n$ have not been applied in the production path, so that the nonterminal symbols $N_{t_1}, \ldots, N_{t_n}$ and the production rules $P_{t_1}, \ldots, P_{t_n}$ are redundant in the production path. Hence we have $G : (S_T, P_T, N_T) = T$. By Definition 10, $t_1, \ldots, t_n \vdash_R F : T$ holds. □